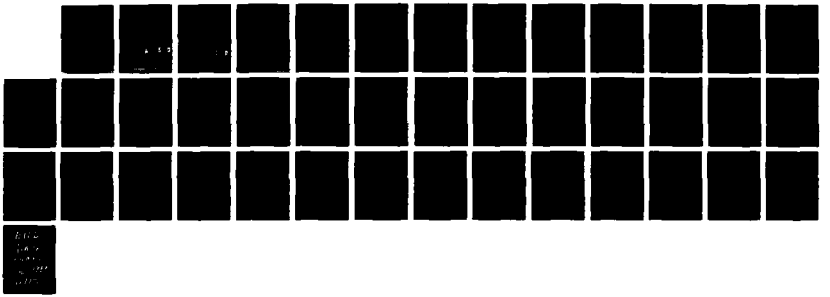
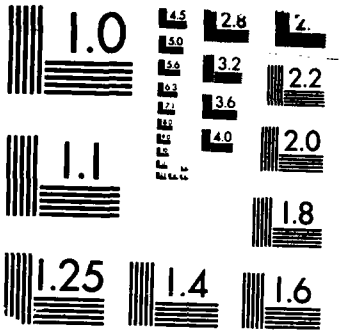
AD-A192 540 SCHEDULING IN REAL-TIME DISTRIBUTED SYSTEMS - A REVIEW (U) MARYLAND UNIV COLLEGE PARK INST FOR ADVANCED COMPUTER STUDIES X YUAN ET AL. DEC 87 UMIACS-TR-87-62 1/1

UNCLASSIFIED N00014-87-K-0241 F/G 12/7 NL





UMIACS-TR-87-62
CS-TR-1955

December, 1987

Scheduling in Real-Time Distributed Systems-A Review†

Xiaoping Yuan
Systems Design and Analysis Group
Department of Computer Science

Satish K. Tripathi and Ashok K. Agrawala
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

UMIACS-TR-87-62
CS-TR-1955

December, 1987

Scheduling in Real-Time Distributed Systems-A Review†

Xiaoping Yuan
Systems Design and Analysis Group
Department of Computer Science

Satish K. Tripathi and Ashok K. Agrawala
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

ABSTRACT

The scheduling problem is considered to be a crucial part in the design of real-time distributed computer systems. This paper gives an extensive survey of the research performed in the area of real-time scheduling, which includes localized and centralized scheduling and allocation techniques. We classify the scheduling strategies into four basic categories: static and dynamic priorities, heuristic approaches, scheduling with precedence relations, allocation policies and strategies.

† This work is supported in part by grants No. 86JJ-HA-40879 OV from Westinghouse Electric Company, and No. N00014-87k-0241 from the Office of Naval Research to the Department of Computer Science, University of Maryland at College Park.

Scheduling in Real-Time Distributed Systems – A Review

Xiaoping Yuan
Satish K. Tripathi
Ashok K. Agrawala

System Design & Analysis Group
Department of Computer Science
University of Maryland
College Park, MD 20742

December, 1987

Abstract

The scheduling problem is considered to be a crucial part in the design of real-time distributed computer systems. This paper gives an extensive survey of the research performed in the area of real-time scheduling, which includes localized and centralized scheduling and allocation techniques. We classify the scheduling strategies into four basic categories: static and dynamic priorities, heuristic approaches, scheduling with precedence relations, allocation policies and strategies.

* This work is supported in part by grants No. 86JJ-HA-40879 OV from Westinghouse Electric Company, and No. N00014-87k-0241 from the Office of Naval Research to the Department of Computer Science, University of Maryland at College Park.

CONTENTS

Note: Because of the limitation of the power of the word processor utilized to edit this paper, some mathematical notations have to be written in English instead of mathematical forms.

1. Introduction

A major application of computers has been to control real-time physical processes. In these applications, the computation required for responding to external events is often periodic and crucial. The responses can not be delayed beyond certain time limits, which are determined by the nature of the physical processes being controlled. Failure to adhere to critical time constraints might bring about catastrophic results.

The actual timing characteristics of software is determined not only by the processor speed, but also by the sharing policy for the scarce resources and the order of task execution. Thus, the scheduling policy of an operating system plays a crucial role in the control of real-time software. The use of distributed multiprocessors for real-time systems further complicates the problem, since the allocation of tasks with timing synchronization and communication constraints to the whole network makes the scheduling much more difficult to implement.

Real-time systems can be divided into two categories: hard real-tme systems and soft real-time systems. In a hard real-time system, an unschedulable task is rejected under any circumstances, while for a soft real-time system, the system works correctly as long as the deadline missing ratio and the expected lateness are below pre-defined thresholds.

The scheduling approaches in distributed systems can be either localized or centralized. *Localized scheduling* is implemented by scheduling tasks in local processors and serves as the basic operation of scheduling in the distributed environment. In *Centralized scheduling*, the schedule of tasks in the whole system is determined by a central scheduler. Allocation techniques can also be divided into two types. One is the *localized allocation*, in which the allocation of tasks is initiated by local processors. Another type is the *centralized allocation*. A centralized allocator pre-determines the location of every task globally. The different

combinations of the above scheduling and allocation techniques are possible and will produce different real-time scheduling schemes.

Table 1 gives a general overview of various papers surveyed here.

The following concepts will help to understand this paper. We say a schedule is *feasible* if all the tasks are scheduled so that none of them misses its deadline. A scheduler is called an *off-line* scheduler if it examines the instances of tasks and creates the scheduling decisions before the run time. A scheduler is called *on-line* if its decisions do not depend on *a priori* knowledge of future requests of tasks and the scheduler chooses a task to run according to the current status of all tasks in the system.

The organization of the paper is as follows. In Section 2, we present the static and dynamic priorites. These are basic scheduling techniques that belong to the localized scheduling techniques.

In Section 3, heuristic methods for scheduling are introduced. The heuristics are more flexible than the static and dynamic priorities in that they can be used to deal with problems such as task scheduling under the conditions of multiple resource requirements, the parallel execution of tasks, and the better utilization requirement of the system. Another important benefit of heuristics is its efficiency in time and space.

In Section 4, we introduce the scheduling techniques under the precedence constraints.

In section 5, we emphasize the allocation policies and strategies including localized and centralized allocation schemes. The benefit of a centralized task allocation scheme is that the system can be well-balanced, because the central processor has global centralized information about every task and processor. However a possible communication overload or a failure of the central processor will cause the whole system to slow down or fail. A localized task allocation scheme is more reliable, but each local processor spends more time on scheduling, allocation, and communication than a centralized one. Therefore, the localized task allocation scheme may reduce the utilization of each processor.

In the final section, we summarize the paper and give the basic requirements and features that a qualified real-time distributed operating system should have.

	Heuristic Approach	Priority Approach	Centralized Allocation	Localized Allocation	Preemption	Precedence Constraints
[1] Cha	x			x		
[2] Che	x			x		x
[3] Cof	x					
[4] Efe	x		x			
[5] Gon	x	x				
[6] Liu	x	x			x	
[7,8] Ma	x		x			x
[9] Mok	x	x			x	x
[10] Mun	x				x	x
[11] Ram	x			x		x
[12] Sha	x	x			x	
[13] Sta	x			x		
[14] Zha	x					
[15] Zha	x				x	
[16] Zha	x			x		

Table1. An Overview

2 Static and Dynamic Priorities

The static and dynamic priorities are the basic scheduling techniques in the real-time scheduling of a local processor. The methods are simple to implement and low in time complexity. Therefore they can be used in on-line schedulers. The operating system requires information on the period, criticalness, execution time, deadline, and initiated time of a task.

2.1 Rate-Monotonic Priority Assignment

Liu and Layland [6] studied the problem of hard real-time scheduling on a single processor with regard to the processor utilization for periodic tasks. It is assumed that the initiated time is the beginning of a period and the deadline is the end of a period. The scheduling algorithms are preemptive and priority-driven. According to [6], a priority-scheduling algorithm is considered to be *static* or *fixed* if priorities are assigned to tasks permanently after the tasks enter a system, and a scheduling algorithm is considered to be *dynamic* if priorities of tasks might be changed by the scheduler during the whole running time of the tasks.

For periodic tasks, one possible static scheduling policy, the *rate-monotonic priority assignment*, is to assign priorities to tasks according to their request rates, independent of their run times. The authors[6] prove that the rate-monotonic priority assignment is optimal such that if there is a feasible static priority assignment for a task set, then the rate-monotonic assignment is also feasible to schedule the task set.

For a case[6] of scheduling two tasks, T_1 and T_2 with periods, P_1 and P_2 respectively, and $P_1 < P_2$, if we let T_1 be the higher priority task according to the rate-monotonic priority assignment, the following inequality must be true to guarantee a feasible schedule.

$$[P_2/P_1]\,C_1 + C_2 \leqq P_2 \qquad (1)$$

(where $[P_2/P_1]$ represents the largest integer less than or equal to P_2/P_1).

If we let T_2 be the higher priority task, then the following inequality must be true,

$$C_1 + C_2 \leqq P_1 \qquad (2)$$

Since by (2), we have

$$[P_2/P_1]C_1 + C_2 \leqq [P_2/P_1]C_1 + [P_2/P_1]\,C_2 \leqq [P_2/P_1]P_1 \leqq P_2$$

(2) implies (1). In other words, whenever the $P_1 < P_2$ and C_1, C_2 are such that the task schedule is feasible with T_2 at a higher priority than T_1, it is also feasible with T_1 at a higher priority than T_2. However, if the task schedule is feasible with T_1 at a higher priority than T_2, it is not true that it is also feasible with T_2 at a higher priority than T_1.

Since the processor utilization factor U is defined to be the fraction of processor time spent on the execution of the task set[6], and C_i/P_i is the fraction of processor time spent on executing task T_i for m tasks, the utilization factor is defined as follows[6]:

$$U = C_1/P_1 + C_2/P_2 + \ldots + C_m/P_m.$$

With the definition of the processor utilization, the authors[6] prove the following schedulability theorem for periodic tasks.

Theorem 1. ([6]) "For a set of m tasks with fixed priority order, the *least upper bound* to processor utilization is $U = m(2^{1/m}-1)$".

It means that as long as the processor utilization is not greater than the least upper bound, the task set is schedulable by one static priority assignment. However, a task set with the processor utilization above the least upper bound can be still schedulable by a static priority assignment, if the periods of the tasks are "suitably" related according to [6].

2.2 Stabilities under the Rate-Monotonic Priority Assignment

Sha, Lehoczky and Rajkumar [12] further developed Liu and Layland's rate-monotonic priority assignment by considering scheduling stability. The *stability* of a system is defined as a scheduling algorithm that is considered to be *stable* if any given set of *critical* tasks is guaranteed to meet all their deadlines even if the processor is overloaded under the assumption that the processor utilization of the task set is below the least upper bound $m(2^{1/m}-1)$. In many cases, some task may be much more critical than the one with a shorter period. A direct application of rate-monotonic priority assignment to the set may result in missing the deadlines of the critical tasks.

A period transformation method has been developed to deal with the stability problem in [12] to use rate-monotonic priority assignment in an unstable system. First, let us look at an example of two tasks from [12]; T_1 = ($P_1 = 12$, $C_1 = 4$, $C_1^+ = 7$) and T_2 = ($P_2 = 22$, $C_2 = 10$, $C_2^+ = 14$) where P_i, C_i and C_i^+ are task i's period, the average and worst case computation times. By theorem 1, these two tasks may be scheduled in the average case, but not in the worst case, since the utilizations of the two tasks in the average case and worst case are 0.79 and 1.2, respectively. The latter exceeds the least upper bound for two tasks: $2(2^{1/2}-1) = 0.82$. Now assume that task T_2 is more critical than task T_1, and that we want to guarantee the deadline of task T_2 even in the worst case condition. We may increase the priority of task T_2 directly. But it will make T_1 miss its deadline even in an average case.

However, this problem can be solved by shortening T_2's period. The *period-shortening approach* is to divide the period of a task into two equal consecutive periods. In the first period, the first half task is executed. The rest of the task is executed at the second period. Therefore, the deadline of the task is still guaranteed to be met. In this example, transform task T_2 to T_2^* = ($P_2 = 11$, $C_2 = 5$, $C_2^+ = 7$). By using the rate-monotonic algorithm, task T_1 and T_2^* can be scheduled even when one of the tasks has the worst case computation time.

It is possible to lengthen the period of task T_1, instead of transforming the period of task T_2 to a shorter one. The *period-lengthening approach* doubles the period of the task, which means that the deadline of the task is postponed to the end of the next period. In order to keep the original frequency of the task executions, add another copy of the task into the system with a different initiated time. The difference is equal to the original period of the task.

For example, task T_1 can be decomposed into two tasks $T_{1,1}$ = ($P_{1,1} = 24$, $C_{1,1} = 4$, $C_{1,1}^+ = 7$) and $T_{1,2}$ = ($P_{1,2} = 24$, $C_{1,2} = 4$, $C_{1,2}^+ = 7$). These two new tasks must be separated by a phase of 12. That is, task $T_{1,1}$ initiates at 0, 24, 48,...etc, while task $T_{1,2}$ initiates at 12, 36, 60, ...etc. The tasks above become schedulable, since postponing the deadline on lengthening the period will automatically reorder the priority of tasks to meet the needs of critical tasks.

A systematic procedure to perform a period transformation[12] is adapted as follows:

```
BEGIN
   IF worst_case_schedulable( Critical_Set ) THEN BEGIN
      Represent the longest period of the critical set by P_c,max and the shortest
      period of the non-critical set by P_n,min;
      IF P_c,max > P_n,min THEN BEGIN
         WHILE worst_case_schedulable( Critical_Set U T_n.min) &
               (P_c.max < P_n,min) DO
            Critical_Set : = Critical_Set U T_n,min;
         WHILE (P_n,min < P_c,max) & (Deadline_Postpone_OK(T_n,min) ) DO
            Lengthen the period of T_n,min;
         WHILE (P_n,min < P_c,max) DO
            Shorten the period of T_c,max;
      END;
      Assign priorities to all tasks according to the rate-monotonic priority
      algorithm;
   END;
END.
```

2.3 Dynamic Scheduling Algorithms

To achieve a better processor utilization, which may have the least upper bound up to be 100 percent, Dynamic priority assignment has been studied by Liu and Layland[6], Mok[9], Gonzalez and Jo[5].

The *deadline driven* scheduling algorithms assign priorities to tasks according to the deadlines of their current requests. One example of this type of scheduling algorithms is the *earliest deadline* algorithm. A task is assigned the highest priority if the deadline of its current request is the nearest. We present two important results of Liu and Layland[6] here.

Theorem 2.([6]) If the earliest deadline scheduling algorithm is used to schedule a set of tasks on a processor, the processor utilization can be 100 percent.

Theorem 3.([6]) For a given set of m periodic tasks, the earliest deadline scheduling algorithm is feasible if and only if the following condition is true,

$$(C_1/P_1) + (C_2/P_2) + \ldots + (C_m/P_m) \leqq 1.$$

The authors[6] also recommend the use of a mixed scheduling algorithm for a practical operating system design. One approach towards the mixed scheduling algorithm lets task 1,2, .. .k, the k tasks of the shortest periods, be scheduled according to the fixed priority rate-monotonic scheduling algorithm, and lets the remaining tasks, task k + 1, k + 2, ..., m, be scheduled according to the earliest deadline scheduling algorithm when the processor is not occupied by task 1, 2, ..., k. The processor utilization of the mixed scheduling algorithm is not as good as the dynamic ones because of the limitation of the static scheduling algorithm. But it may be appropriate to many applications, especially for those interrupt services that need immediate execution.

Mok[9] has introduced another dynamic priority approach, the *least slack* algorithm or *minimum laxity* algorithm. This algorithm denotes the remaining computation time of a ready task at time t by c(t) and its current deadline by d(t), and defines the slack of the task at time t by max{d(t)-t-c(t), 0}. The slack or laxity is the maximum time the scheduler can delay the task before it is going to miss its deadline. The least slack algorithm is that the scheduler should select the task with the least slack to run. According to the proof in [9], the algorithm is a totally on-line optimal algorithm.

Theorem 4. ([9]) The least slack algorithm can be used as a totally on-line optimal run-time scheduler under the assumption that the scheduler can choose to preempt a task by any other task.

COMMENTS

Under one-processor resource scheduling, the earliest deadline algorithm outperforms the least slack algorithm in the point of view that some unnecessary task preempting overheads are avoided.

In [9], it is also observed that there are an infinite number of totally on-line optimal schedulers. Any combination of the earliest deadline and the least slack algorithm can be used in a run-time scheduler to minimize task preempting

overheads. An important result is obtained if we take mutual exclusion constraints into consideration. The mutual exclusion limits the preemption.

Theorem 5. ([9]) If there are mutual exclusion constraints in a system, it is impossible to find a totally on-line optimal run-time scheduler because of the limitation on the preemption.

A dynamic priority scheduling algorithm that takes user dissatisfaction of response time into consideration has been introduced in [5] for soft real-time system. In the scheduling, the algorithm increases the priority level of every task throughout its stay in the system at a rate indicated by its allowance for response time as soon as the task enters into the system.

The system is monitored for each pre-defined interval. The scheduling algorithm is preemptive. The response time, D, of a task is assumed to have discrete value, that can also be taken as a deadline. According to [5], at the time t, the priority level q_k of a task k, k = 1, 2,.., is defined by

$$q_k(t) = \begin{cases} (b/D_k)\cdot(t-r_k) & t \leqq r_k + D_k \\ b + a\cdot(t-r_k-D_k) & t > r_k + D_k \end{cases}$$

where *a* and *b* are pre-defined constants and the task has entered the system at time r_k with allowance D_k.

It can be seen from the above expressions that a task continuely increases its priority starting with zero and reaches a fixed level *b* by its deadline time. A difference rate *a* is used for tasks that wait beyond their deadlines. This is a soft real-time system that tries to minimize the mean user's waiting time and dissatisfaction time. The dissatisfaction time is defined as an *excess time* of deadline [5].

A simulation based on the processor utilization factor is described in [5]. A comparison with three other scheduling policies (First Come First Served, Shortest Remaining Service Time First, and Earliest Delay Time First) shows that this policy to minimize the user dissatisfacton time gives a better result under different utilization factors.

3. Heuristic Approaches

An optimal scheduling solution is a time-consuming NP-hard problem for a multiprocessor distributed system. It may be possible to use heuristics to improve the efficiency of the scheduler, although some optimality cannot be achieved. Heuristics are informal, and try to find a path in a scheduling search tree in an efficient way of including plausible nodes and excluding implausible nodes in the tree to avoid the exponential search in the worst case. The algorithms discussed in this section are all localized scheduling algorithms.

3.1 Multiple-Resource Requirement without Preemption

The scheduling problem with resource constraints and without preemption was studied in [14]. The authors not only considered the computation times and deadlines, but also took into account that a task can request any number of local resources, including CPU, I/O devices and files, in a way that can also be extended to scheduling of tightly-coupled distributed real-time systems. A tightly-coupled distributed system is a parallel computer system of multiple CPUs with one shared memory.

This model is a loosely-coupled distrubuted system with homogeneous nodes. Each node contains a set of distinct resources, R_1, R_2, ... R_r. A resource can be shared serially by tasks. A task T is not only attributed with computation time C_T, deadline d_T, but also with the resource requirement (that can be represented as a 0/1 vector with dimension r).

According to [14], the local scheduler uses a vector EAT to indicate the earliest available times of resources:

$$EAT = (EAT_1, EAT_2, ..., EAT_r)$$

where EAT_i is the earliest time when resource R_i will become available. Each time the partial schedule is extended, a new task T is added into the original schedule S, and EAT is updated by taking into account the resource requirement and completion time of the new task.

The schedule is represented by a search tree. The tree is expanded by adding a task into an old tree to create a new partial schedule. The expansion process is implemented by selecting the most promising task from all the remaining tasks by using a heuristic function. The search tree then, is expanded one more level by using a new vertex to indicate the new scheduled task. In order to make the algorithm computationally tractable, only one vertex is selected at each level of the search tree.

At each level of the search tree, the scheduler calculates a vector called DRDR, the *dynamic resource demand ratio*, that indicates the fraction to which the remaining tasks will use the resource [14]:

$$DRDR = (DRDR_1, DRDR_2, ..., DRDR_r)$$

where $DRDR_i$ is defined as:

$$DRDR_i = \frac{SUM(C_T, T \text{ remains to be scheduled and uses } R_i)}{MAX(d_T, T \text{ remains to be scheduled and uses } R_i) - EAT_i}$$

where i = 1, ..., r.

The constraints of the search for a *strongly feasible* schedule are

1) $DRDR_i \leqq 1$ for i = 1, ... r; and

2) All of its immediate extensions are feasible.

COMMENTS

We find that the constraint DRDR for a strongly feasible schedule can be further improved with the same computation time to catch an infeasible schedule at an earlier time than the above one . The new algorithm for testing resource j is as follows.

```
SUM = 0;
FOR i = 1 TO n DO BEGIN   /* n is the number of tasks to be scheduled */
    SUM = SUM + Ci;
    IF SUM / (di - EATj) > 1 THEN
        RETURN (Not Guaranteed)
END {FOR}.
```

If a partial schedule is not strongly feasible because condition 1 or 2 fails, it is very clear that all further extensions will fail to meet the constraints. Given the

above constraints, the search should be confined only to those subtrees which possibly lead to a feasible schedule.

If there are some tasks which cannot be scheduled locally, they are sent to other nodes in the network to be rescheduled. This allocation method will be fully discussed in Section 5. The pseudo code for the algorithm in [14] is adapted below.

```
PROCEDURE scheduler( task_set; schedule )
BEGIN
    schedule : = empty;
    WHILE NOT empty( task_set ) DO BEGIN
        calculate ST for each task in task_set;
        calculate DRDR;
        IF not feasible( schedule ) THEN
            return( not guaranteed )
        ELSE BEGIN
            calculate New_EAT for each task in the task_set;
            apply heuristic function H to each task in the task_set;
            let T be the task with the minimum value of function H;
            task_set : = task_set - {T};
            schedule : = schedule U {T};
            EAT : = New_EAT(T);
        END
    END
    return( guaranteed );
END.
```

In the algorithm, the heuristic function H is defined as follows.

$$H_T = W_1*X_1 + W_2*X_2 + W_3*X_3$$

where T is a task, W_1, W_2, and W_3 are weights that are tunable variables, and the following three statements hold.

(1). X_1 equals the laxity of the task T, and is given by

$$X_1 = d_T - (<\text{Task-T-Start-Time}> + C_T)$$

(2). $X_2 = C_T$, the computation time of the task T.

(3). X_3 takes into account the resource requirements of the task T as well as the resource utilization. X_3 is directly proportional to the time at which a resource is idle when the task is scheduled next, and inversely proportional to the time and number of possible parallel task executions.

The authors of [14] have done a simulation by using the above algorithm with a success ratio varying from 72.5% to 81% depending on other factors such as deadline tightness. After extending the basic algorithm by limited-backtracking, the authors found the success ratio can go up to 99.5% in their simulation model. Interested readers are referred to their work[14].

The time complexity of the heuristic scheduling algorithm is $O(rn^2)$, which is a reasonable cost compared with exhaustive searching of a scheduling tree. The algorithm can generally be used for off-line scheduling and batch processing.

3.2 Mutiple-Resource Requirement with Preemption

A further development of the scheduling algorithm in the last section is introduced by the same authors [15]. The model here is still a loosely-coupled distributed system with homogeneous nodes. Each node in the network contains a set of distinct resources, $R_1, R_2, ..., R_r$. The preemption is utilized to achieve a better schedule in the new strategy.

Zhao *et al.* [15] take a different approach from their earlier paper[14] by choosing one more *slice* (instead of one more task) as the next extended vertex of the search tree. The slice is constructed with one *primary* task and several *secondary* tasks. The primary task is selected according to timing consideration only: minimum laxity algorithm. Then the secondary tasks are selected by using a combination of various heuristics, which will be shown in the end of this section.

A schedule S for a set of preemptable tasks consists of a sequence of *slice* S_k, $k = 1,...$ NS. NS stands for the number of slices in the schedule S. A slice S_k is associated with a *slice start time* SST_k, a *slice length* SL_k. A slice consists of a subset of tasks which can run concurrently. A task T_j will be executed during a slice S_k if T_j

belongs to S_k. A task is preempted instead of finished between slice S_k and S_{k+1} if the task is executed in slice S_k, but not completed in that slice, and also not in the next slice S_{k+1}.

Under the slice terminology, a schedule is feasible, if for all j, j = 1, ..., NS,

$$MAX(SST_k + L_k : T_j \text{ belongs to } S_k) \leqq D_j.$$

In [15], the following formula shows how to select the length of each slice. EST is defined as the *earlist start time* of a new slice. A slice length SL_k is defined as,

$$SL_k = \min(\min(C_j' : T_j \text{ belongs to } S_k), \min(d_h - C_h' - EST : T_h \text{ does not belong to } S_k \text{ and } C_h' > 0))$$

where the first term presents that the slice length is not longer than the minimum remaining processing time C_j' of tasks in the slice, and the second term specifies that the length should be chosen to be not longer than the minimum of the laxities of the remaining tasks. Another concept used in [15] is the *minimum resource demand ratio*, MRDR. The MRDR indicates the minimum fraction to which the remaining tasks will use resources.

$$MRDR = (MRDR_1, MRDR_2, ..., MRDR_r)$$

where $MRDR_i$ is defined as:

$$MRDR_i = \frac{SUM(C_j' : T_j \text{ requires } R_i \text{ exclusively}) + Max(C_j' : T_j \text{ requires } R_j \text{ in shared mode})}{MAX(d_j : T_j \text{ requires } R_j \text{ and } C_j' > 0) - EST}$$

The constraints for a *strongly feasible* schedule are

1). $MRDR_i \leqq 1$ for i = 1,... , r; and

2). the latest slice has a length > 0.

COMMENTS

The similar improvement as the one mentioned in the last section for DRDR, can be made towards the calculation of MRDR in order to catch an infeasible schedule earlier than the above equation.

Since we have described the basic properties of the algorithm, we omitted the outline of the algorithm. However it should be noted that the heuristic function H in the algorithm can take into consideration not only the minimum resource idle time, and the maximum parallelism mentioned in the last section, but also the following simple heuristics[15]:

(1) minimum deadline first :

$H(T_j) = D_j$;

(2) minimum laxity first :

$H(T_j) = d_j - C'_j$;

(3) minimum remaining processing time first:

$H(T_j) = C'_j$;

(4) maximum remaining processing time first:

$H(T_j) = -C'_j$;

(5) minimun shared resource requirements first:

$H(T_j)$ = number of resources T_j uses in shared mode;

(6) maximum shared resource requirements first:

$H(T_j)$ = -number of resources T_j uses in shared mode;

(7) minimum or maximum resource utilization first;

H has been discussed in Section 3.1 (X_3 of heuristic function).

The authors [15] simulated the scheduling process with different combinations of the above heuristics and limited backtracking. The success ratio varies from 85% to 98%, depending on the schedule length, the tightness of deadline and other tunable variables.

4. Scheduling with Precedence Relations

This section examines the topic of scheduling with precedence constraints, since tasks in a real-time system usually cooperate with one another and have relationships like ordering, mutual exclusion, inter-task or interprocess communication.

4.1 The Actual Initiated Time and Deadline

For a set of tasks $\{T_1, T_2, ..., T_n\}$, assume that the knowledge of initiated time r_i, deadline d_i and precedence relations $(T_i->T_j)$ of every task in the task set is known *a priori*. Here $T_i->T_j$ can be read as T_i precedes T_j or T_j follows T_i. Any unspecified initiated time can be regarded as the absolute time 0 of the scheduling block [0, L], where L is the MAX(d_i: for all i). A simple and effective method [9] to calculate the actual initiated times and deadlines of tasks with precedence relations is adapted as follows.

BEGIN

Sort the tasks generated in [0, L] in a forward topological order;

Initialize the initiated time of task T_i in [0, L] to r_i;

Revise the initiated time in a forward topological order by the formula: [r_i = MAX(r_i, $\{r_j + c_j: T_j->T_i\}$)] where T_i, T_j are tasks in [0,L], and r_i, r_j and c_j represent the current request-time of T_i, the current initiated time of T_j and the computation time of T_j;

Sort the tasks in a reverse topological order;

Initialize the deadline of task T_i in [0, L] to d_i;

Revise the deadline in a reverse topological order by the formula: [d_i = min(d_i, $\{d_j-c_j:T_i->T_j\}$)], where d_i, d_j and c_j represent the current deadline of T_i, the current deadline of T_j and the computation time of T_j;

END.

The effect of the above procedure is to assign to each task in [0, L] an initiated time r_i which is the earliest time at which it can be scheduled, and a deadline d_i which is the latest time by which it must be completed if their partial ordering is to be maintained. Any scheduling algorithm can use the above information to schedule a task set with precedence relations.

4.2 Minimizing Execution Time under Precedence Relations

Muntz and Coffman [10] studied the preemptive scheduling of real-time tasks on multiprocessor systems with precedence constraints. In their model, each computation is associated with a graph, G, whose nodes correspond to the set of tasks. There is a directed arc from the node representing task T_i to the node representing T_j if and only if T_i->T_j. A weight is associated with each node of the graph as the execution time of the corresponding task. Each of the computation graphs is an acyclic, weighted and directed graph.

A variation in [10] from the traditional models is that the k processors in a system comprise a certain amount of computing capability rather than being discrete units, with the assumption that preemption cost and communication time are negligible compared with the computation costs. Thus, this computing capability can be assigned to tasks in any amount between zero and one unit of a processor. If computing capability a, $0 < a \leqq 1$, is assigned to a task, then the computation time of the task is increased by a factor of 1/a naturally. The following is the definition of their disciplines.

General Scheduling (GS) discipline:

The amount of computing capability between zero and one is dynamically assigned to a task according to a heuristic algorithm.

Preemptive Scheduling (PS) discipline:

Only one unit computing capability can be assigned to a task at one time.

Example.

One possible *General Scheduling (GS) discipline* for the graph in Fig. 1 adapted from [10] is shown with the equivalent of *Preemptive Scheduling (PS) discipline*.

It is shown in [10] that GS is equivalent to PS. Actually they are mutually transformable. But since GS is a better model to describe the scheduling, we will concentrate on the GS for the scheduling discussion.

Here we present an algorithm from [10] for constructing an optimal GS (PS) for any number of processors when the computation graph is a rooted tree. Before the algorithm is presented, the following terminology is helpful. Consider a tree T, and two nodes T_i, T_j that belong to the tree such that T_j->T_i. The distance between T_i and T_j is the sum of the computation times of all the nodes (including both T_i and T_j)

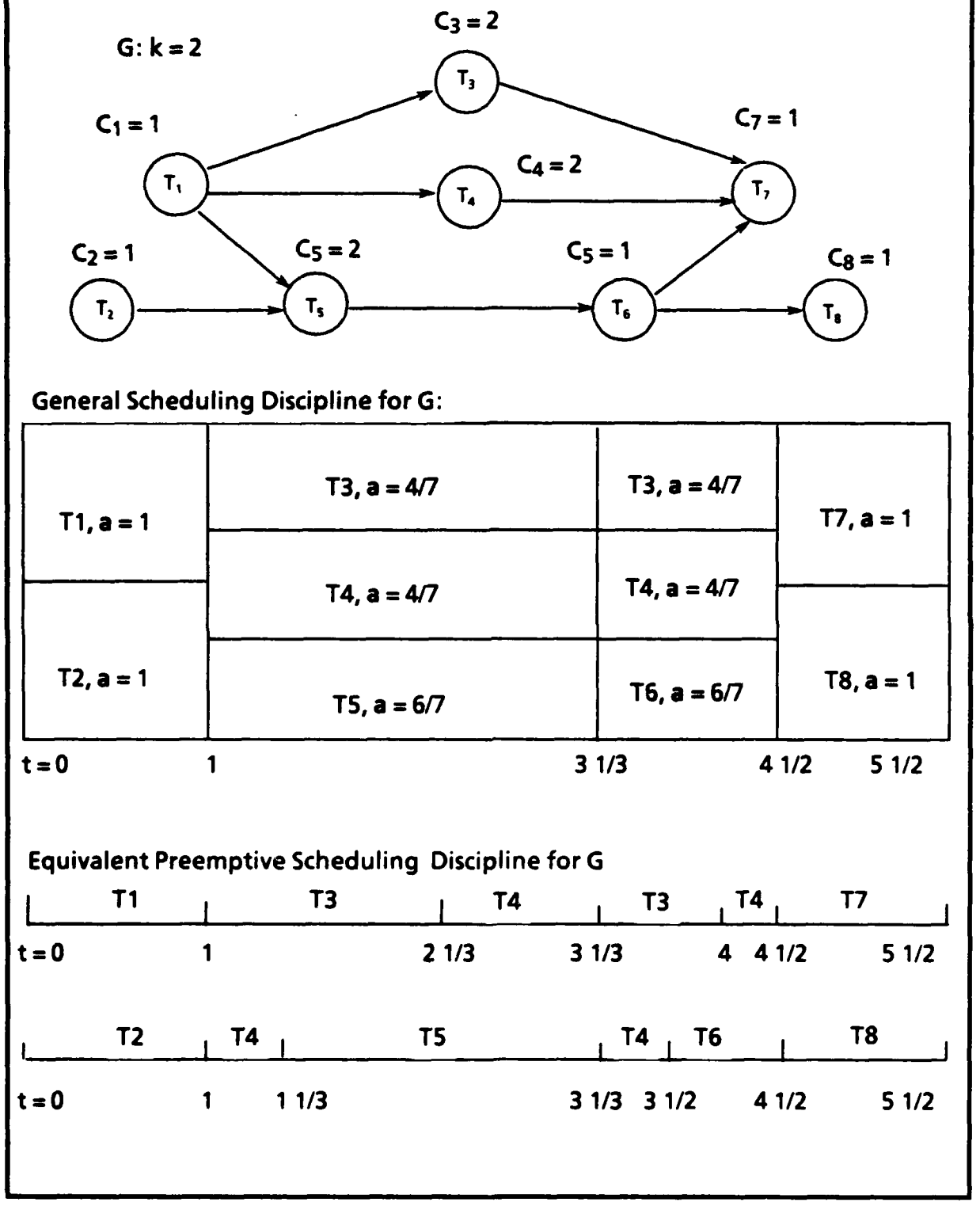

G: k = 2
C3 = 2
T3
C1 = 1
C7 = 1
T1
C4 = 2
T4
T7
C2 = 1
C5 = 2
C5 = 1
C8 = 1
T2
T5
T6
T8
General Scheduling Discipline for G:
T1, a = 1
T3, a = 4/7
T3, a = 4/7
T7, a = 1
T4, a = 4/7
T4, a = 4/7
T2, a = 1
T5, a = 6/7
T6, a = 6/7
T8, a = 1
t = 0
1
3 1/3
4 1/2
5 1/2
Equivalent Preemptive Scheduling Discipline for G
T1
T3
T4
T3
T4
T7
t = 0
1
2 1/3
3 1/3
4
4 1/2
5 1/2
T2
T4
T5
T4
T6
T8
t = 0
1
1 1/3
3 1/3
3 1/2
4 1/2
5 1/2

Fig. 1

in the unique path connecting T_i and T_j. The executions are supposed to begin with the leaves of the tree. The distance from node T_j to the root of the tree is called the *height* of the node T_j. Finally, at any point of the execution in a task tree, the executable nodes (the node without precedence) are the initialized nodes of the tree remaining to be processed.

Assume T be a tree and k be the number of available processors. Then an algorithm for constructing an optimal General Scheduling discipline for tree T is as follows.

Algorithm. ([10]) Under the assumption that the heights of all tasks are known *a priori.* Assign one processor (a = 1) to each of the k nodes farthest from the root of T. If there is a tie among b nodes (because they are at the same level) for the last *m* machines ($m \leqq b$), then assign m/b of a processor to each of these b nodes. Each time either one of two situations described below occurs, reassign the processors to the tree that remains to be computed according to this rule. The two situations are:

Event 1. A node in the tree T is completed.

Event 2. A point is reached where, if the present assignment is continued, some nodes would be computed at a faster rate than others that are farther from the root of the tree T.

The following example can further explain the algorithm.

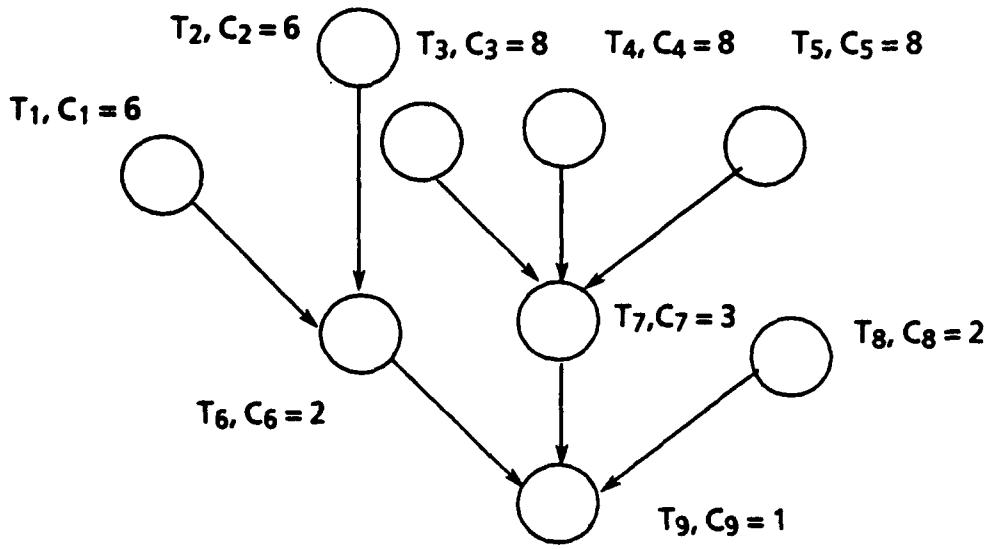
Example:

For the computation tree shown in Fig. 2(a) and k = 3 processors, the schedule for the tree T is illustrated in Fig. 2(b). At t = 3 there is an occurence of event 2, since the executions of T_3, T_4 and T_5 reaches the height of T_1 and T_2. At this point the three processors are shared equally by nodes T_1, T_2, T_3, T_4 and T_5. The whole example is shown in the Fig. 2.

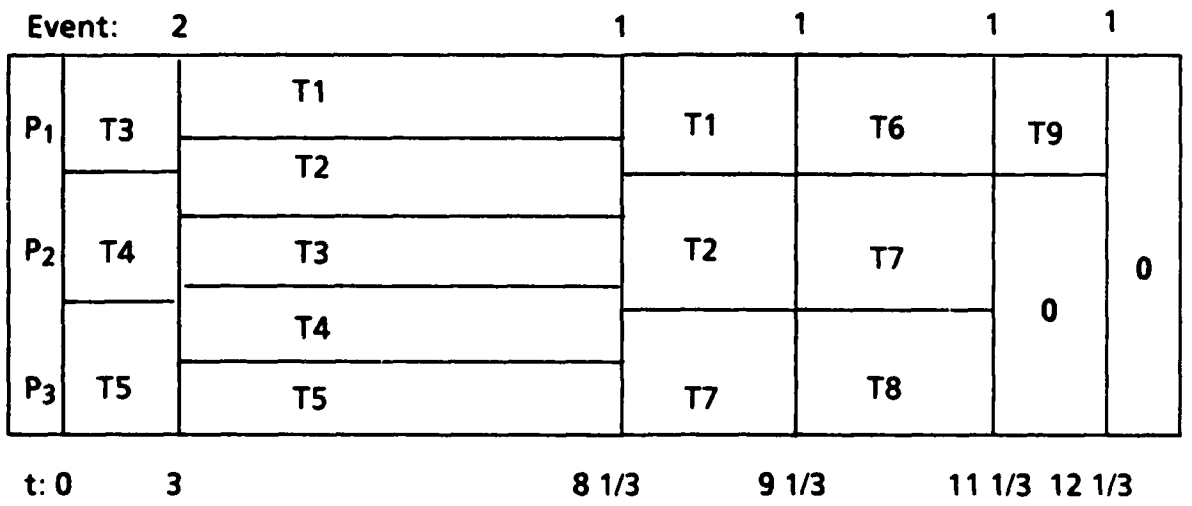
A very interesting proof that the algorithm is optimal is given in Muntz and Coffman[10] under real-time environment with precedence relations. Interested readers are referred to their paper.

5. Scheduling and Allocation Policies and Strategies

In order to take full advantage of multi-processors which are connected by a network to run tasks, it is crucial to have a good policy of allocation and scheduling



(a) k = 3



(b) General Scheduling

Fig. 2

to ensure that when a task is in execution at any node in the network, it still adheres to constraints such as precedence relations, deadlines, and inter-task communications on different processors.

5.1 Clustering and Local Scheduling

Cheng *et al.*[2] have introduced a dynamic scheduling and allocation strategy for groups of tasks with precedence constraints in a distributed hard real-time system.

The model of [2] is for a homogeneous distributed network in which the aperiodic-task groups may arrive any time. Tasks in each group are interrelated by arbitrary precedence constraints. Each task in a group must finish by one group deadline, and each task may have multiple predecessor tasks and multiple successor tasks. When a predecessor task finishes, it sends an enabling message, and output data, to each of its successor tasks. A successor task is enabled if it receives all the enabling signals and input data it needs. In hard real-time systems, the time spent in communication between tasks is accounted for explicitly in scheduling each group.

The approach is different from others, in that the clustering technique is applied into the localized scheduling and allocation. The purpose of the task clustering is to maximize the parallel computation, reduce the communication cost for the scheduling, allocation and final execution of the task groups and take advantage of precedence relations among tasks in the scheduling and allocation.

A task group is decomposed into a set of clusters. With the clusters, the following work can be done. First, try to schedule as many clusters locally as possible. Second, for those clusters, which cannot be scheduled locally, send them to other nodes. The allocation techniques are discussed in the next section.

The outline of the scheduling algrithm owed to Cheng *et al.*[2] is shown as follows. The focussed addressing and bidding of clusters belongs to the allocation techniques.

```
/* Decide the schedulability of a task group */
BEGIN
    IF the entire group is scheduled locally THEN Return(Success);
    ELSE        /* Distribute it in the network to be scheduled */
```

PREPROCESSING:

Estimate the number of network nodes with sufficient surplus to take part in distributed schduling;

LOCAL CLUSTER SCHEDULING:

Attempt to schedule each cluster locally;

/* The following part belongs to allocation strategies */

FOCUSSED ADDRESSING AND BIDDING:

FOR each cluster that cannot be scheduled **DO**

Find a focussed node for the cluster and send the cluster to the node directly;

FOR the clusters for which focussed nodes are not available **DO**

Send a request for bid messages to other nodes;

Send each cluster to the best bidder nodes;

FOR each cluster for which bidder nodes are not available

invoke a compression algorithm and attempt to reschedule the cluster locally;

END.

5.2 Localized Allocation – Focussed Addressing and Bidding

The *Focussed Addressing and Bidding scheme* actually performs the function of distributing local tasks or clusters of tasks to other nodes. Localized allocation has been introduced in [2, 13, 16].

When a task or a cluster of tasks cannot be guaranteed at the local node, the original node, the node will try to send the tasks to other lightly-loaded nodes by interacting with the schedulers on other nodes in an attempt to find a node that has sufficient surplus to guarantee the tasks. This interaction is based on a scheme that combines the focussed addressing and bidding. We use a task instead of a cluster of tasks in the following description. The transformation is obvious. The algorithm of the focussed addressing and bidding for distributed scheduling is as follows.

A. The Focussed Addressing Scheme

The focussed addressing scheme is as follows.

BEGIN

Choose candidates for a focussed node according to the local information of surplus of the network;

IF there is any candidate for the focussed node **THEN BEGIN**

randomly choose one of the candidates to be the focussed node, say node j;

send the task to node j;

store the task in bidder's queue of tasks for which Request For Bids have to be sent out (indicate that bids should be returned to the focussed node j);

END ELSE

store the task in bidder's queue of tasks for which Request For Bids have to be sent out (indicate that bids should be returned to the requesting node);

END IF

END.

In the scheme, every node keeps track of the surplus of other nodes which is updated periodically. Focussed addressing utilizes network-wide surplus information to reduce overheads incured by bidding in determining a good node to send a task to, and works as follows. Before sending request for bids, a node uses the surplus information about the other nodes in the network to determine if a particular node has surplus that is significantly greater than the computation time of the new task that could not be guaranteed locally.

Meanwhile, the bidding process is invoked when communication with the focussed node is in progress. The bids will return to the focussed node if it receives the task; otherwise, the bids return to the requesting node.

B. The Bidding Scheme

In [2, 13 and 16], the main functions of the bidding scheme on a node are: sending out the request for bids for those tasks that cannot be guaranteed locally,

responding to the request for bids from other nodes, evaluating bids, and processing the arriving bidded task.

a). Request For Bids (RFB)

When a task cannot be guaranteed locally, a decision is made as to whether to transmit a request for bids. This decision is based on calculating the earliest possible response time and deadline for response. The earliest possible response time takes into account the fact that a request for bids is handled by a remote node's bidder task and that two-way communication is involved with the bidder request. Such nodes can be identified by using the local surplus information of all nodes.

b) Bidding in Response to RFB

When a node receives a request-for-bid message, it calculates a bid for the task. The bid depends on the free time of each resource required by the bidded task. If the bid is big enough to guarantee the task, it is sent to the focussed addressing node, if there is one. Otherwise, the bid is sent to the original node which issued the request-for-bid message.

c). Bid Processing

When a node receives a bid for a given task, and the bid is higher than a certain threshold, the node awards the task to the bidding node immediately. All other bids for this task, that arrived earlier or may arrive later, are discarded. If all the bids, that have arrived, for a given task are lower than the threshold, the node postpones making the awarding decision until L(T), the latest bid arrival time of the task. At time L(T), the task will be awarded the highest bidder if any, where the bid should be above a threshold; otherwise, the task can not be guaranteed. All the bids that arrive later will be discarded.

d). Response to the Arriving Bidded Task

Once a task is sent to a node, the receiving node treats it as a task that has arrived at the node and takes action to guarantee it. If the task cannot be guaranteed, the node can request for bids and determine if some other node has

the surplus to guarantee it. However, since the task was sent to the best bidder and the task's deadline will be closer than before, the chances that there is another node with surplus big enough for the task are small. Hence, if the best bidder can not guarantee the task, it sends the task to the second best bidder, if any. Otherwise, the task is also rejected. The location of the second best bidder can be sent to the best bidder with the task from the bid-requesting node.

5.3 Bidding under Precedence Relations

In the above focussed addressing and bidder schemes, the way in which one should handle the tasks with precedence constraints is not discussed. In [12], another bidding algorithm is given with the consideration of precedence relations, under the assumption that clocks are synchronized. Their algorithm is as follows.

BEGIN

Attempts to guarantee T_1 locally, assuming a start time of D_{max} + Max(Comm_Delay);

/* where D_{max} is the latest deadline of all preceding tasks, and Max(Comm_delay) is the maximum time required for the communication delay.) */

IF this local guarantee is successful **THEN Return**(SUCCESSFUL);

IF task T_1 can not be guaranteed at its arrival site A **THEN**
node A broadcasts a request for bids with an indication that bids should be returned, not to itself, but the node, say B, containing the preceding task with the latest deadline. Also, node A sends the task T_1 itself to node B;

WHEN the task arrives at node B = > attempt to guarantee it at that node.
IF this is not successful, **THEN** the local guarantee algorithm attempts to modify D_{max} to D_{max}', so that $D_{max}' < D_{max}$ and D_{max}' is still the latest deadline and the task with D_{max}' remains guaranteed, and task T_1 can be guaranteed;

IF task C is still not guaranteed **THEN** the node B waits for returning bids and chooses the best bidder as in the normal bidding process;

END.

5.4 Centralized Allocation

Although various papers have been written on the centralized allocation issue, we still do not find any of them using the real-time scheduling approach. Here we introduce two of them which might help to design a real-time centralized allocation strategies.

In [4, 7, 8] , heuristic models are considered to solve the task distribution or allocation problems. Their models are centralized and off-line schedulers. The one in [7, 8] has been applied successfully to scheduling a group of real-time tasks, although scheduling approach in this model is not a real-time one. We first outline Efe's work[4] as follows.

Under the distributed environment with a group of tasks to be scheduled, the heuristic approach in [4] can be expressed by a two-phase algorithm:

```
BIGIN
/* Phase 1:   Find the best assignment */
   FOR i = 2 TO N DO BEGIN          /* N is the number of processors */
      Form i module clusters by using a Module Clustering Algorithm;
      Assign the above i module clusters to separate processors to maximize the
      load balancing;
      IF This assignment achieves the best maximum load balancing result THEN
         Substitute the previous one by this new assignment;
   END;
/* Phase 2:   Under the best assignment, balance the load of the network */
   WHILE The load-balancing constraints is not satisfied DO BEGIN
      Identify the overloaded and underloaded processors;
      Reassign some modules from overloaded processor to underloaded
      processors;
   END
END.
```

where the module clustering algorithm is designed to maximize the parallel computation, minimize the communication cost.

But Ma *et al.*[7, 8] took a different approach. Their work is to set up constraints for the search tree of scheduling and allocation by using branch and bound algorithm to find a suboptimal schedule.

A. Information

The allocation scheme makes use of previously known information on the tasks and the network. The useful informaton on tasks includes coupling factors among tasks, task size, and the number of enablements of each task. The coupling factor between tasks is a measure of the number of data units transferred from one task to another. The tasks with coupling factors are in a *thread* which is defined as a group of interrelated tasks.

The useful information on the network is the interprocessor distance and processor constraints. The interprocessor distance is the physical distance between two processors.

B. Constraints

The authors take several constraints into the allocation model. The constraints are used in the search for an allocation scheme by using the *branch* and *bound* algorithm. Some of them are listed below.

(1) The *memory attribute* which is represented by

$$SUM_i(M_i * X_{ik}) \leqq S_k$$

where M_i is the amount of memory required by task i, and S_k represents the memory capability at processor k. The space required by the tasks concurrently residing in one memory can not exceed the processor memory capacity.

(2) The *task preference matrix,* which indicates that certain tasks can only be executed in the specified processor, is represented by an $n \times m$ matrix P (m is the number of tasks and n is the number of processors), where $P_{ik} = 0$ means that a task i cannot be assigned to processor k; and $P_{ik} = 1$, otherwise.

(3) The *task exclusive matrix,* which indicates mutually exclusive tasks, is represented by an $m \times m$ matrix E, where $E_{ij} = 1$ implies that task i and j cannot be assigned to the same processor; and $E_{ij} = 0$, otherwise.

C. Cost Constraints

The cost function calculated as the sum of the interprocess communication cost and the processing cost. Interprocess cost is a function of both task coupling factors and interprocessor distances. The coupling factor U_{ij} is the number of data units transferred from task i to task j. The interprocessor distance d_{kl} is a distance-related communication cost associated with one unit of data transferred from processor k to processor l. If task i and j are assigned to processor k and l respectively, the interprocessor cost is ($U_{ij} * d_{kl}$).

Processing cost q_{ik} represents the cost to process task i on processor k. The assignment variable is defined as follows:

$$X_{ik} = \begin{cases} 1 & \text{if task i is assigned to processor k} \\ 0 & \text{otherwise} \end{cases}$$

The total cost for processing the tasks is stated as

$$SUM_i\ SUM_k(Wq_{ik}X_{ik} + SUM_l\ SUM_j(U_{ij} * d_{kl})\ X_{ik} * X_{jl})$$

The normalization constant W is used to scale the processing cost and the interprocessor communication cost to make up the difference in measuring units.

The total cost will be calculated every time that the scheduler determines which processor the next task should be in. The usage will be described in the next algorithm.

D. The Algorithm

Once all of the above information is available, the algorithms[7, 8] are based on *a branch* and *bound* (*BB*) method, which was defined by Kohler and Steiglitz in [3]. To employ the BB technique, the allocation problem is represented by a search tree. The allocaton decision represents a branching at the node corresponding to the

given task. Consider a problem of allocation of m tasks among n processors. Starting with task 1, each task is allocated to one of the n processors subject to the constraints imposed on the relations of tasks and processors. That is, one branch from n candidates is selected at every extention step.

By the definition of the branch and bound algorithm(BB), the BB method consists of nine rules: (B, S, F, D, L, U, E, BR, RB).

(1). *Branching rule B* selects a branch (processor) for a given node (task). It defines the scheme for generating sons of each node.

(2). *Selection rule S* selects the next branching node to expand from the currently active node. This can be done by choosing a task according to the order of each thread.

(3). *Characteristic function F* eliminates nodes known to have no complete solution from the set of feasible schedules. This is determined by checking the preference matrix P for task *k* and processor *i*.

(4). *Dominance relation D* defines the relations on the set of partial schedules and will be used by rule E to eliminate nodes from the search tree before extending them.

(5) *Lower bound function L* calculates a cost lower bound to each partial schedule.

(6) *Cost upper bound U* of a complete schedule is known at the beginning of the algorithm and will be used in rule E.

(7) *Elimination rule E* uses rules D, L and U to eliminate new nodes by checking the task exclusive matrix for task *k*. If tasks *l* and *k* which are mutually exclusive and task *l* has already been allocated to processor *i*, then the branch *i* for node *k* is eliminated. Rule E also compares the partial cost L with the cost upper bound U. If L is greater than U, the schedule cannot be improved. Hence, branch i for k is eliminated.

(8). *Termination rule BR* derives the possible optimal cost from an acceptable schedule. The Rule BR terminates the algorithm when all possible paths have been investigated and no feasible schedule is achievable. The task set will be rejected.

(9). *Resource bound vector RB* checks the resource requirements and capacity. If the cumulative size of requirements for a resource such as the memory exceeds the resource capacity, the corresponding branch will be elimilated.

The authors applied the algorithm to a real-time project, which has a lower interprocess communication cost with similar load balance effects compared to the allocation based on the experience.

5.5 Sender-Initiated vs. Receiver-Initiated Allocation

The local allocation under deadline constraints by the comparison of sender-initiated and receiver-initiated approaches, in a soft real-time environment, is studied in [1].

According to [1], in a soft real-time distributed system, a task can be in one of the following three states: *blessed*, *to-be-late*, and *late*. A task is *blessed* if it will finish within its deadline. If the task can only meet its deadline by migrating the task to other nodes, this taks is called as a to-be-late task. A task enters the *late* state when it is clear that it will definitely miss its deadline.

A. Receiver-initiated deadline scheduling

At each time a task is complete, the load of the processor is checked to determine whether the processor is underloaded. When the number of tasks left in the local processor is smaller than a pre-defined threshold N_1, the processor is considered as underloaded. (Comments: We think the major factors for determining the status of the underload should be the total computation time of all tasks in the processor, besides the number of tasks in the system). When an underloaded state is entered, the allocation module starts polling other processors to ask for jobs. Two-phase polling is used in order to give higher priority to the most urgent tasks. A "to-be-late" task is the most urgent because the task can still finish in time if the task can be migrated to a processor that can guarantee it. A "late" task is the second urgent since the migration of the "late" task may reduce the overload of the current processor and avoid more tasks entering the "late" state. A "blessed" task is the least urgent since it can complete in time at the current processor.

In the first phase, the polling processor tries to find a task in the state of "to-be-late" or "late". Then in the second phase, a "blessed" task is sought to reduce the probability of deadline-missing in the future, if the polling processor is still underloaded.

In order to prevent the polling message from overloading the communication, the limitation of reinvocation of the polling is needed.

B. Sender-initiated deadline scheduling

The sender-initiated deadline scheduling is similar to the approach studied in [2], which is invoked when a local overloaded situation occurs. A processor is considered overloaded if one or both of the following conditions are met[1]:

(1) There is at least one of the tasks in the processor in a "to-be-late" or "late" state.

(2) There are more than N_2 tasks in the processor. N_2 is a pre-defined threshold.

When a task arrives, the local scheduler checks the local processor load. If the processor is in an overloaded state, the scheduler selects a task candidate and initiates a migration try on every other processor in the network. The heuristic rules for selecting a migraton candidate are listed below[1]:

(1) Select among the "to-be-late" tasks with the earliest deadline,

(2) Choose a "late" tasks,

(3) Otherwise, select a "blessed" task that has the longest processing time with the condition that the task will not miss its deadline due to the delay incured in a task migration.

The receiver accepts a migration request only if both of the following conditions are met[1]:

(1) the load is lower than the threshold N_1;

(2) the candidate can be guaranteed.

In [1], a discrete event simulator was built to evaluate the performance of the two algorithms. Each algorithm was simulated for 9000 time units under different task arrival rates. Fig. 3 from [1] shows their simulation results, which presents the deadline miss ratio, the percentage of tasks that do not meet their deadlines, of the two algorithms as a function of task arrival rate. When the rate is less than 0.75, the sender-initiated algorithm (Send-Init) performs better than the receiver-initiated scheduling algorithm (Rcv-Init). However, when the arrival rate exceeds 0.75, the Rcv-Init algorithm becomes superior. Fig. 3 also compares the dynamic scheduling with the *isolated* scheduling under which processors do not transfer tasks.

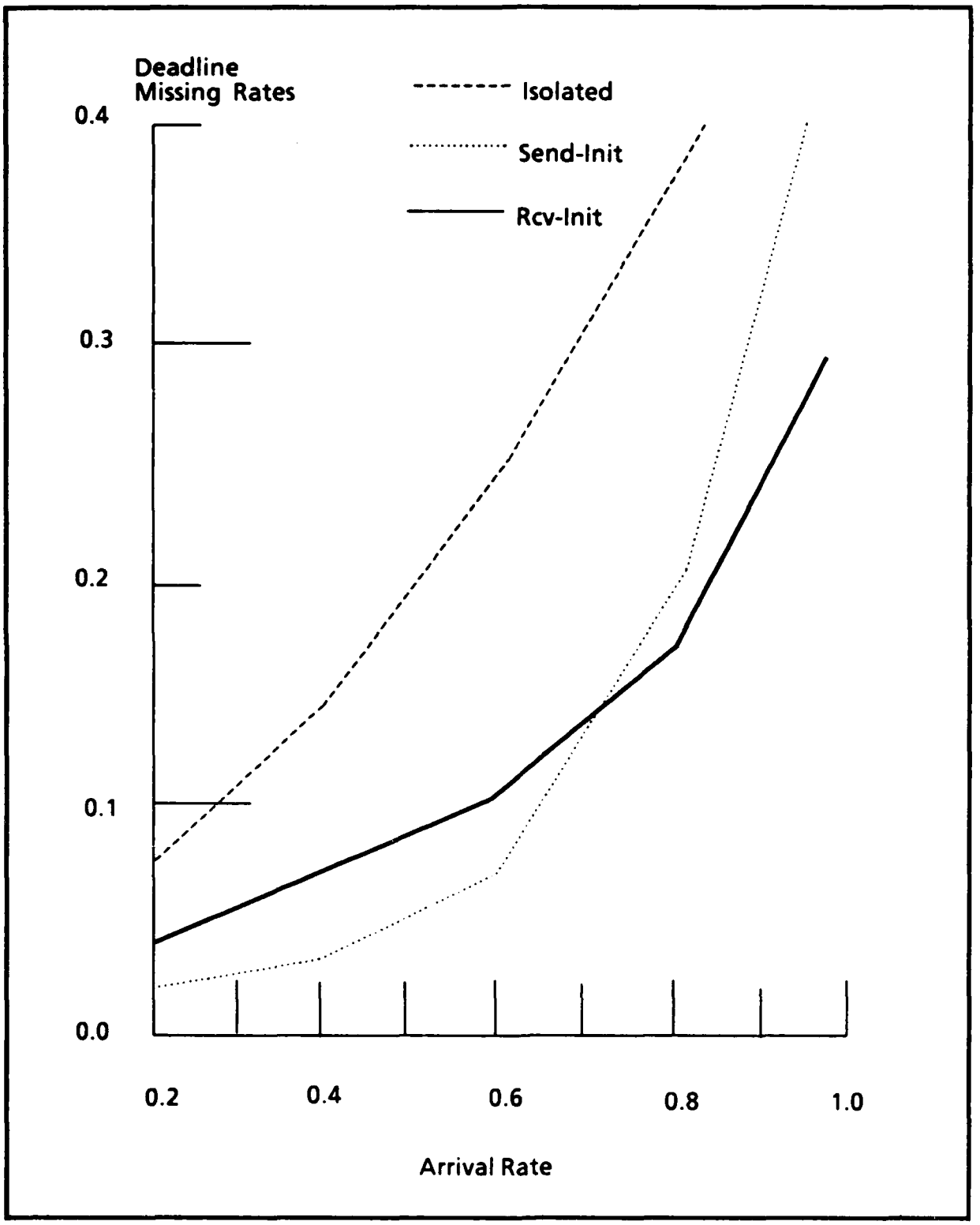


Fig. 3 The simulation results[1]

6. Conclusion

From the above description, we think that the problem of local real-time scheduling is basically solved with the help of different methods that include static and dynamic priority algorithms, heuristics for multiple-resource constraints, stability algorithms, clustering methods and a General Scheduling discipline.

However, although different models and approaches have been established to deal with *distributed* real-time systems, none of them is overwhelmingly superior and satisfactory, and a lot of improvement can still be made. It means that the real-time scheduling under the distributed environment is still an open problem.

A practical and well-formed real-time distributed operating system should include at least the following features:
(1). Each task is attributed with a deadline, computation time, period (or the least time between two invocations) and initiated time.

(2). The mutual exclusion such as critical section, semophore and monitor among the tasks can be achieved.

(3). The intertask communication on one or more processors should be guaranteed.

(4). Each task is also restricted by precedence relations explicitly.

(5). Preemption is allowed.

(6). The computing system consists of a loosely-coupled distributed system.

Acknowledgements

The authors thank C.R. Prasad, S.-T. Levi for their valuable comments and suggestions in the preparation of this paper, and N. Muckenhirn for her careful English editing.

7.References

[1] Chang, H-Y., Livny, M., Distributed scheduling under deadline constraints: a comparison of sender-initiated and receiver-initiated approaches, *Proc. IEEE Real-Time Syst. Symp*. Dec. 1980.

[2] Cheng S., Stankovic, J.A., Ramamritham, K., Dynamic scheduling of groups of tasks with precedence constraints in distributed hard real-time systems, *Proc. IEEE Real-Time Syst. Symp*. Dec. 1986.

[3] Coffman, E.G. Jr., Ed, *Computer and Job-Shop Scheduling Theory*, New York, Wiley 1976.

[4] Efe, K., Heuristic models of task assignment scheduling in distributed systems, *IEEE Computer*, June 1982.

[5] Gonzalez, C., Jo, K.Y., Scheduling with deadline requirements, *Proc. of 1985 ACM Annual Conference* (Denver, Colorado) Oct. 1985.

[6] Liu, C.L., Layland, J., Scheduling algorithm for multiprogramming in a hard real-time environment, *J.ACM*, Vol 20, Jan 1973.

[7] Ma, R.P., Lee, Y.S., Tsuchiya, M., Design of task allocation scheme for time critical applicaton, *IEEE Proc. Real Time Syst. Symp.*, Miami Beach, FL, DEC.1981.

[8] Ma, R.P., Lee, Y.S., Tsuchiya, M., A task allocation model for distributed computing systems, *IEEE Trans. Comput*. Vol C-31, Jan. 1982.

[9] Mok, A., *Fundamental Design Problems for the Hard Real-Time Environment*, MIT Ph.D. Dissertation, May 1983.

[10] Muntz, R.R., Coffman, E.G., Preemptive scheduling of real-time tasks on multiprocessor systems, *J.ACM*, Vol 17, Apr.1970.

[11] Ramamritham, K., Stankovic, J.A., Dynamic task scheduling in distributed hard real-time systems, *IEEE Software*, Vol 1, July 1984.

[12] Sha, L., Lehoczky, J., Rajkumar, R., Solutions for some practical problems in prioritized preemptive scheduling, *Proc. IEEE Real-Time Syst. Symp*. Dec. 1986.

[13] Stankovic, J.A., Ramamritham, K., Cheng, S., Evaluation of flexible task scheduling algorithm for a distributed hard real-time system, *IEEE Trans. Comput.* Vol C-34, Dec. 1985.

[14] Zhao, W., Ramamritham, K., Stankovic, J.A., Scheduling tasks with resource requirements in a hard real-time system, *IEEE Trans. Software Eng.* Vol SE-13. No.5 May 1987.

[15] Zhao, W., Ramamritham, K., Stankovic, J.A., Preemptive scheduling under time and resource constraints, *IEEE Trans. Comput.* Vol C-36, No.8, Aug. 1987.

[16] Zhao, W., Ramamritham, K., Stankovic, J.A., Distributed scheduling using bidding and focused addressing. *IEEE Proc. Real-Time Syst. Symp.* Dec. 1985.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UMIACS-TR-87-62 CS-TR-1955	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION University of Maryland 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) Department of Computer Science University of Maryland College Park, MD 20742	7b. ADDRESS (City, State, and ZIP Code) 800 North Quincy Street Arlington, VA 22217-50000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-87K-0241
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)

Scheduling in Real-Time Distributed Systems - A Review

12. PERSONAL AUTHOR(S)

Xiaoping Yuan, Satish K. Tripathi and Ashok K. Agrawala

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	December, 1987	37

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The scheduling problem is considered to be a crucial part of the design of real-time distributed computer systems. This paper gives an extensive survey of the research performed in the area of real-time scheduling, which includes localized and centralized scheduling and allocation techniques. We classify the scheduling strategies into four basic categories: static and dynamic priorities, heuristic approaches, scheduling with precedence relations, and allocation policies and strategies.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

END

DATE

FILMED

6-1988

DTIC